[1]

Optimización por Colonia de Hormigas: Aplicación al Problema del Agente Viajero

J. A. Carmona, *Estudiante, UDEA*

***Abstracto*—Se presenta la implementación del algoritmo de Optimización por Colonia de Hormigas (Ant Colony Optimization, ACO en Inglés) para aplicar en la solución al Problema del Agente Viajero (Travelling Salesman Problem, TSP) a través del lenguaje de programación de Python; Se generan varias instancias al problema con el fin de hallar los parámetros de la colonia que mejor se adapta a los casos presentados en el artículo. Se compara el modelo metaheurístico aplicado en la solución con el modelo óptimo GLPK que nos entrega la solución exacta.**

***Índices de términos*— Ant Colony Optimization, ACO, Travelling Salesman Problem, TSP, Metaheuristic, GLPK, python3.**

I. Introducción

El objetivo del problema del agente o vendedor viajero (TSP) es encontrar la ruta con menor costo posible empezando desde una ciudad y visitando todas las ciudades del conjunto con la restricción de que sólo se puede visitar una vez durante todo el recorrido para terminar nuevamente en la ciudad de origen.

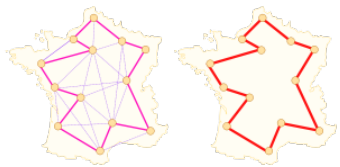

Figure 1: Ciclo Hamiltoniano

Es un caso especial de los problemas de optimización donde las variables a controlar son discretas y las soluciones son un conjunto de permutaciones o series de combinación.

II. Modelo de optimización: problema del agente viajero

En teoría de grafos, el Problema del Vendedor Viajero (TSP) se puede representar como un problema de combinatoria a través de un grafo $G=(X,E,W)$, que contiene un conjunto de aristas con un coste en longitud asociado a cada una, y se trata de hallar el ciclo Hamiltoniano que presenta el menor costo en la suma de sus aristas que lo componen, donde X, E y W, representa el conjunto de nodos, enlaces y su costo asociado, respectivamente.

$$X=(c_1,c_2,...,c_n)n\geq 3$$
$$E=e_{ij}\big|_{\square} c_i,c_j\in X$$
$$W=w_{ij}\big|_{\square} w_{ij}\geq 0\wedge w_{ii}=0,\forall i,j\in\{1,2,...,n\}$$

Para nuestro caso, denotaremos la distancia o el coste d_{ij} de la arista entre las ciudades $c_i=(x_i,y_i)$ y $c_j=(x_j,y_j)$.

De forma explícita, las ciudades pueden representarse como una red de puntos en un espacio bidimensional, y su distancia puede ser calculada por medio de la distancia euclídea entre ambas ciudades.

$$d_{ij}=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$
Ecuation 1

Por lo tanto, esta distancia será simétrica, es decir, tendrá el mismo corte ir de c_i a c_j que ir desde c_j a c_i.

En Teoría de la Complejidad, es considerado como un Problema NP-completo el cual crece de forma exponencial con la entrada del problema, y su entrada es representada por el número de nodos (ciudades o sitios) del grafo.

Por lo tanto, no es factible intentar resolver el problema utilizando fuerza bruta a través de todas las combinaciones posibles, debido a los costos actuales en computación. Es necesario utilizar otras implementaciones que permitan llegar a una solución lo suficientemente optimizada sin demandar demasiados recursos en computación.

La siguiente metaheurística implementada ha sido inspirada en la forma en como las colonias de hormigas encuentran y recolectan su alimento.

III. Aplicación de la metaheurística: optimización por colonia de hormigas

La Optimización por Colonia de Hormigas hace parte de los algoritmos de inteligencia de enjambre inspirados por el comportamiento social de las hormigas al hallar y recolectar alimento. Esta especie coopera mediante un medio indirecto de comunicación (feromonas), realizando movimientos en el espacio de decisión (permutaciones).

[1]Johany Andrés Carmona C., estudiante ingeniería de Telecomunicaciones, Departamento de Electrónica y Telecomunicaciones, Universidad de Antioquia, MED 050012 COL (e-mail: johany.carmona@udea.edu.co)

A modo inicial, estos insectos exploran aleatoriamente su entorno en busca de alimento, una vez encontrado, evalúan su cantidad y calidad, regresando una parte del alimento hallado.

Mientras la hormiga regresa al nido, dejará una cantidad de feromona sobre el camino que dependerá de la cantidad y calidad del alimento encontrado, la cual servirá de guía para que el resto de las hormigas encuentren los alimentos.

Como inicialmente no hay presencia de feromonas en los diferentes caminos alternos, la elección del camino será realizada de forma aleatoria.

El camino utilizado por la hormiga puede estar lleno de obstáculos, por lo cual, no es necesariamente el camino más corto, de esta manera las hormigas transitarán por diferentes caminos hacia las distintas fuentes de comida, evaluando su cantidad y calidad, creando un efecto acumulativo de feromonas sobre los caminos más cortos desde su nido hacia las fuentes de alimento, que al final permitirá converger rápidamente desde una estrategia de diversificación hacia una estrategia de intensificación.

Dado un conjunto n de ciudades con distancia d_{ij} entre ellas, las hormigas se distribuirán de forma aleatoria entre las ciudades del conjunto. Cada hormiga visitará la próxima ciudad (vértice) de acuerdo a las marcas de feromona dejada por las hormigas en cada camino (arista) y su nivel de visibilidad.

- Al principio, si la distancia entre dos caminos es la misma, las hormigas tenderán a escoger el camino con menor cantidad de feromona.

Adicionalmente, existen dos principales diferencias entre las hormigas reales y las hormigas artificiales que se usarán en la aplicación del Problema del Agente Viajero.

- Las hormigas artificiales tienen memoria y pueden recordar las ciudades que han visitado, de esta manera no volverán a ser seleccionadas en el siguiente paso.
- Las hormigas artificiales no se encuentran completamente ciegas, ya que conocen las distancias entre dos ciudades y tienden a preferir elegir aquellas que se encuentrán más cerca de su posición actual.

a) Función de probabilidad de paso en cada nodo para cada hormiga

De esta manera, podemos representar la probabilidad de que la ciudad j es visitada por la hormiga k después de haber visitado la ciudad i, de la siguiente forma:

$$p_{ij}^{k}=\begin{cases}\dfrac{\left[\tau_{ij}\right]^{\alpha}.\left[\eta_{ij}\right]^{\beta}}{\sum_{s\in allowed_k}\left[\tau_{is}\right]^{\alpha}.\left[\eta_{is}\right]^{\beta}}, si\ j\in allowed_k\\ 0, si\ j\notin allowed_k\end{cases}$$

Ecuation 2

- τ_{ij} representa la intensidad de la feromona del camino desde la ciudad i hasta la ciudad j.
- α es un parámetro de control que nos permite medir la influencia de la intensidad de la feromona.
- $\eta_{ij}=\frac{1}{d_{ij}}$ indica el nivel de visibilidad de la ciudad j desde la ciudad i.
- β es un parámetro de control para regular la influencia del nivel de visibilidad entre ciudades.
- $allowed_k$ contiene el conjunto de ciudades que no han sido visitadas aún por la hormiga k.

Las hormonas son optimizadas de tal forma que el camino más corto obtiene más cantidad de feromona que el camino más largo.

Si hacemos $\alpha=0$, las ciudades más cercanas en cada paso serán las que tendrán más probabilidad de ser seleccionadas. (Algoritmo voraz clásico).

Si hacemos $\beta=0$, únicamente intervendrá la feromona, lo que puede llevar a peores soluciones sin posibilidad de mejora.

b) *Actualización del nivel de intensidad de la feromona entre caminos transitados por la colonia*

Es importante establecer un parámetro que permita decaer la influencia de la feromona sobre los caminos que dejan de ser transitados en cada iteración.

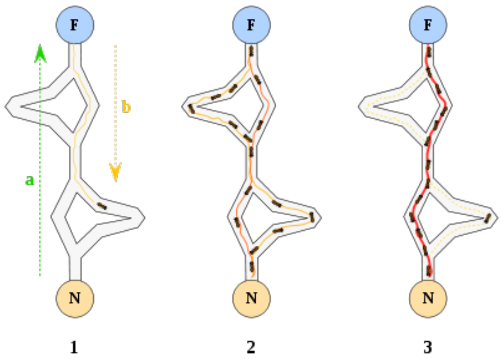

Figure 2: Actualización feromona en la colonia

La actualización del nivel de intensidad de la feromona en todo el grafo se realiza luego de que cada hormiga completa las iteraciones para todas las ciudades del conjunto y obtiene el tour o ruta completa. Se puede calcular a través de la siguiente expresión:

$$\tau_{ij}(t+1) = \rho . \tau_{ij}(t) + \Delta\tau_{ij}$$

Ecuation 3

- Donde $\rho \in [0,1]$ representa el factor de evaporación o reducción del nivel de feromona en el grafo.

$$\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^{k}$$

Ecuation 4

- $\Delta\tau_{ij}$ representa el incremento total del nivel de feromona sobre el camino (i, j) realizado por todas las hormigas de la colonia.

$$\Delta\tau_{ij}^{k} = \left\{ \begin{array}{c} \frac{Q}{L_k}, if\ ant\ k\ travels\ on\ edge\ (i,j) \\ 0, otherwise \end{array} \right\}$$

Ecuation 5

- $\Delta\tau_{ij}^{k}$ indica el incremento del nivel de feromona sobre el camino (i, j) realizado por la hormiga k. Q se refiere al nivel de intensidad de la feromona. L_k representa la distancia recorrida por la hormiga k.

Por ejemplo, para cada generación o iteración, si la hormiga *k* ha realizado el recorrido $T_k(t)$, de longitud total $L_k(t)$, para cada par $(i, j) \in T_k(t)$, se puede hacer un depósito de feromona de $\Delta\tau_{ij}^{k}(t) = \frac{Q}{L_k(t)}$

c) Pseudocódigo de Optimización por Colonia de Hormigas (ACO) para el Problema del Agente Viajero (TSP).

En cada iteración del algoritmo, cada hormiga construirá una solución al problema a través de un grafo. Donde cada arista representa las posibles opciones que el insecto puede tomar, en base a la siguiente información.

- Información heurística: Mide la preferencia heurística que tienen las hormigas para moverse de una ciudad a otra.
- Información de los rastros de feromonas artificiales: Mide la deseabilidad de visitar una ciudad *i* después de visitar una ciudad *j*, imitando el comportamiento social de las hormigas en busca de alimento.

En forma general, el algoritmo usado para la solución del problema formulado del Agente Viajero (TSP) usando la Optimización por Colonia de Hormigas (ACO) se describe de la siguiente forma.

Table 1: Pseudo-code of ACO for TSP

```
Initialize
For t=1 to iteration number do
   For k=1 to l do
      Repeat until ant k has completed a tour
         Select the city j to be visited next
         With probability p_ij given by Eq. (2)
      Calculate L_k
   Update the trail levels according to Eqs. (3-5)
End
```

Entre las variantes para mejorar la eficiencia del algoritmo, para este caso, se enfocó en tres esquemas de actualización de la feromona:

- (0): Sistema de Ranking, donde cada hormiga deposita feromona de forma proporcional a la calidad de la solución hallada.
- (1): Sistema de Colonias, como mecanismo análogo al observado en la naturaleza, en el cual todas las hormigas dejan la misma cantidad de feromona, independiente de la solución.
- (2): Sistema Elitista, Añade feromona en cada paso a las aristas del mejor recorrido hallado hasta el momento.

IV. Implementación: modelo desarrollado en lenguaje de programación e intérprete python

Se analiza las clases y métodos principales para el funcionamiento del Modelo de Optimización por Colonia de Hormigas (ACO).

Se muestra los métodos desarrollados para la función de probabilidad de paso en cada nodo para cada hormiga; y el método de actualización del nivel de intensidad de la feromona entre caminos transitados tanto para la hormiga, como para toda la colonia.

Table 2: Ant Colony Optimization Classes

...	...
...	Colony
Graph	Ant
ACO	

Para el desarrollo del modelo ACO fue necesario el desarrollo de dos subclases: Graph y Ant, respectivamente. Del mismo modo, estas subclases dependen de otras clases o estructuras de datos para poder procesar los elementos del problema y los parámetros de la metaheurística que se usarán en la ejecución del algoritmo, y las cuales no hacen parte de la temática principal del presente artículo (ver Anexo).

a) Clase Graph

Se encarga de generar y almacenar toda la información de los costos entre ciudades y el nivel de feromona presente entre todos los caminos de la red de ciudades.

Filename: graph.py

```
import random
import math

"""
Graph contains information about cost matrix and
pheromoneSpace status.
"""
class Graph(object):
    def __init__(self, locations: list, GLPK: bool = False):
        """
        :param costMatrix: distances between all sites on the
graph.
        :param pheromoneSpace: pheromone value for each
trail on the graph.
        :param GLPK: indicates if the graph will be used to
generate a GLPK format data file, that can be used to solve
using GLPK model (tsp.mod)
        """
        self.GLPK = GLPK
        self.totalSites = len(locations)

        #Initialize matrix with distances between sites.
        self.costMatrix = [ [ self.cost(locations[i] , locations[j])
if i != j else 0 for j in range(self.totalSites) ] for i in
range(self.totalSites) ]

        #Initialize matrix with a residual pheromone value for
each status on the graph.
        self.pheromoneSpace = [[1/self.totalSites**2 if
self.totalSites != 0 else -1]*self.totalSites]*self.totalSites

    #Generate Euclidean cost from bidimentional space.
    def cost(self, originLocation: tuple, destinyLocation:
tuple):
        cost = math.sqrt((originLocation[0] -
destinyLocation[0]) ** 2 + (originLocation[1] -
destinyLocation[1]) ** 2)
        return cost if not(self.GLPK) else round(cost)
```

b) Clase Colony

Estructura o clase de datos que almacena los parámetros de la colonia que pertenece la hormiga.

Filename: colony.py

```
class Colony(object):
    def __init__(self, alpha: float, beta: float, rho: float, Q: int,
scheme: int):
        """
        :param alpha: Pheromone parameter to regulate the
influence of the intensity of pheromone trails between sites.
        :param beta: Local node parameter to regulate the
influence of the visibility between sites.
        :param rho: Pheromone evaporation factor to regulate
the reduction of pheromone.
        :param Q: Pheromone deposit factor
        :param scheme: pheromone update procedure.
            0: ant-ranking system
            1: ant-colony system
            2: ant-elitist system
        """
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        self.scheme = scheme
```

c) Clase Ant

Clase que contiene todo el proceso sobre cómo las hormigas pueden actuar manipulando la matriz de costos y el espacio de feromonas de acuerdo a su recorrido ejecutado.

Filename: ant.py

```
import random
from graph import Graph
from colony import Colony
"""
Ant contains information about how ants can act
manipulating the cost matrix and pheromoneSpace status
based on its colony parameter.
"""
class Ant(object):
    def __init__(self, colony: Colony, graph: Graph):
        """
        :param colony: Contains the colony parameters of the
ant.
        :param graph: Contains the graph of the problem.
        :param totalCost: Total cost acumulated on the ant tour.
        :param tabu: Contains the cities visited for the ant.
        :param deltaPheromone: Contains the pheromone
changes generated by each Ant over each edge on the graph.
        :param allowedSites
        :param currentSite
        :param eta: Contains the visibility level for each edge on
the graph.
        """
```

```
        self.colony = colony
        self.graph = graph
        self.totalCost = 0.0
        self.tabu = []
        self.deltaPheromone = []
        self.allowedSites = [i for i in range(graph.totalSites)]
        self.eta = [[1 / graph.costMatrix[i][j] if i != j else 0 for j
in range(graph.totalSites)] for i in range(graph.totalSites)]
        originSite = random.randint(0, graph.totalSites - 1)
        self.tabu.append(originSite)
        self.currentSite = originSite
        self.allowedSites.remove(originSite)

    def nextMovement(self):
        denominator = 0
        for site in self.allowedSites:
            denominator +=
self.graph.pheromoneSpace[self.currentSite][site] **
self.colony.alpha * self.eta[self.currentSite][site] **
self.colony.beta
        probabilities = [0]*self.graph.totalSites

        for site in self.allowedSites:
            probabilities[site] =
self.graph.pheromoneSpace[self.currentSite][site] **
self.colony.alpha * self.eta[self.currentSite][site] **
self.colony.beta / denominator

        #Select next site according by probability each site.
        nextSite = 0
        randomSite = random.random()
        for site, probability in enumerate(probabilities):
            randomSite -= probability
            if randomSite <= 0:
                nextSite = site
                break
        self.allowedSites.remove(nextSite)
        self.tabu.append(nextSite)
        self.totalCost += self.graph.costMatrix[self.currentSite]
[nextSite]
        self.currentSite = nextSite

    def updateDeltaPheromone(self):
        self.deltaPheromone = [[0] * self.graph.totalSites] *
self.graph.totalSites
        for step in range(len(self.tabu)-1):
            i = self.tabu[step]
            j = self.tabu[step + 1]
            if self.colony.scheme == 0: #ant-ranking system
                self.deltaPheromone[i][j] = self.colony.Q /
self.totalCost
            elif self.colony.scheme == 1: #ant-colony system
                self.deltaPheromone[i][j] = self.colony.Q
            elif self.colony.scheme == 2: #ant-elitist system
                self.deltaPheromone[i][j] = self.colony.Q /
self.graph.costMatrix[i][j]
```

d) Clase ACO

Clase final que contiene el modelo metaheurístico de Optimización por Colonia de Hormigas enfocado en aplicar técnicas de diversificación e intensificaciones con la finalidad de obtener la mejor solución en un tiempo razonable para el problema del Agente Viajero (TSP).

Filename: aco.py

```
import random
from graph import Graph
from colony import Colony
from ant import Ant
"""
Ant Colony Optimization (ACO) contains the Metaheuristic
model in charge of apply diversification and intensifications
techniques to obtain the best solution on the Travelling
Salesman Problem (TSP).
"""
class ACO(object):
    def __init__(self, iterations: int, totalAnts: int, alpha: float,
beta: float, rho: float, Q: int, scheme: int):
        """
        :param iterations
        :param totalAnts
        :param colony
        """
        self.iterations = iterations
        self.totalAnts = totalAnts
        self.colony = Colony(alpha, beta, rho, Q, scheme)

    #Updating the Pheromone Space where is all the
pheromone deposited for each ants of the colony.
    def updatePheromoneSpace(self, graph: Graph, ants: list):
        #Evaporating Pheromone Space
        for i in range(graph.totalSites):
            for j in range(graph.totalSites):
                graph.pheromoneSpace[i][j] *= self.colony.rho
                #Adding delta Pheromone from each ant of the
colony.
                for ant in ants:
                    graph.pheromoneSpace[i][j] +=
ant.deltaPheromone[i][j]

    def solveModel(self, graph: Graph):
        """
        :param graph
        """
        bestCost = float('inf')
        bestSolution = []
        for iteration in range(self.iterations):
            ants = [Ant(self.colony, graph) for i in
range(self.totalAnts)]
            for ant in ants:
                for i in range(graph.totalSites - 1):
                    #Next movement according visibility allowed
nodes.
                    ant.nextMovement()
                #End movement from end node to start node.
                ant.totalCost += graph.costMatrix[ant.tabu[-1]]
```

```
[ant.tabu[0]]
            if ant.totalCost < bestCost:
                bestSolution = ant.tabu
                bestCost = ant.totalCost
            ant.updateDeltaPheromone()
        self.updatePheromoneSpace(graph, ants)
    return bestSolution, bestCost
```

V. Análisis y resultados

Para la evaluación del rendimiento del modelo, se eligieron conjuntos de 13, 21, 34, 55 y 89 puntos distribuidos de forma aleatoria entre el rango de [0 , 100] tanto para el eje x e y, del plano cartesiano.

Además, se generó un archivo de texto plano con los datos expresados en formato GLPK para poder ser utilizados para comparar con el modelo óptimo GLPK.

a) Conjunto Problema: 13 Ciudades

Table 3: Conjunto Datos Generados, 13 Ciudades

Executed Command	
python3 main.py generateSites GLPK data13 13 0 100 0 100	
OUTPUT	
GLPK data file	./data/GLPK/data13.dat
ACO data file	./data/data13.dat

1) Resultados Modelo GLPK

Table 4: Rendimiento Modelo GLPK, 13 Ciudades

Executed Command	
glpsol -m tsp.mod -d ./data/GLPK/data13.dat -o ./data/GLPK/data13-Output.dat	
OUTPUT	
GLPK Output file	./data/GLPK/data13-Output.dat
Cost	383
Route	[1, 5, 6, 11, 12, 2, 9, 8, 3, 4, 7, 10, 13]
Runtime	0.5 s

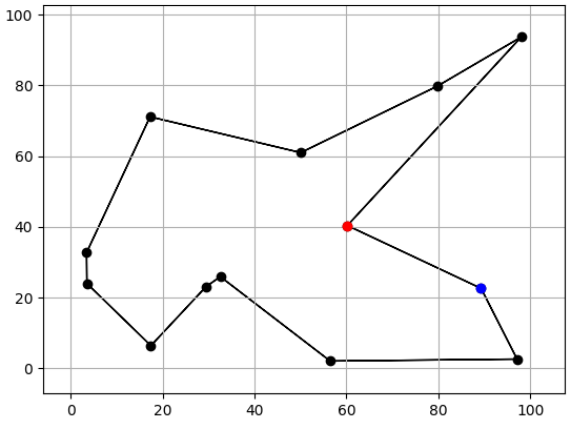

Figure 3: Ciclo Hamiltoniano Modelo GLPK, 13 Ciudades

2) Resultados Modelo ACO

Se realizó diversificación en los parámetros de la colonia para encontrar los parámetros de las cinco colonias más aptas para trabajar el conjunto de datos generado previamente.

Para esto, se generó 144 colonias con parámetros generados de forma aleatoria, con tres variaciones en el número de hormigas en el rango [21,89]; con dos variaciones en el parámetro alpha y beta en el rango [1,13]; con dos variaciones en el parámetro rho de evaporación de feromona en el rango [0.21,0.89]; y dos variaciones en el parámetro Q de intensidad de feromona en el rango [1,3] con un máximo de 8 iteraciones o generaciones por colonia.

Table 5: Resultados Diversificación Colonias, 13 Ciudades

Executed Command								
python3 main.py solveMultipleModels GLPK data13 8 3 21 89 2 1 13 2 1 13 2 0.21 0.89 2 1 3								
OUTPUT								
Results			./data/data13-RESULTSx8.dat					
Log			./data/data13-LOGx8.dat					
cost	colony	Ants	alpha	beta	rho	Q	scheme	Runtime
383	24	46	11.08	5.6	0.5	1	0	0.33
383	34	46	11.08	5.6	0.75	3	1	0.33
383	37	46	11.08	12.82	0.5	1	1	0.34
383	21	46	10.25	12.82	0.75	3	0	0.34
383	4	46	10.25	5.6	0.5	3	1	0.34

Luego se ejecutó el modelo implementando uno de los parámetros de la mejor colonia, y usando 144 iteraciones o generaciones.

Table 6: Rendimiento Modelo ACO, 13 Ciudades

Executed Command	
python3 main.py solveModel data13 144 46 10.25 5.6 0.5 3 1	
OUTPUT	
Colony	4
Iterations	144
Cost	383
Route	[12, 2, 9, 8, 3, 4, 7, 10, 13, 1, 5, 6, 11]
Runtime	4.74 s

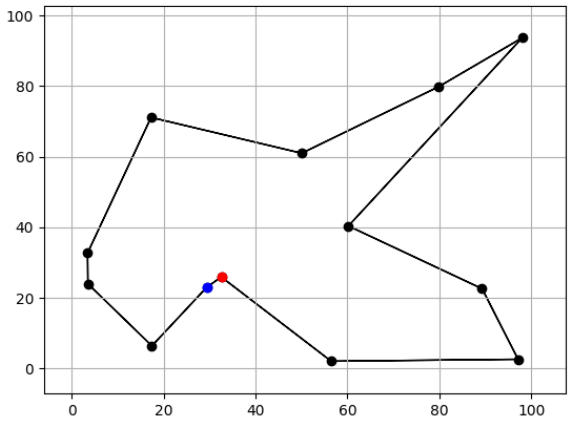

Figure 4: Ciclo Hamiltoniano Modelo ACO, 13 Ciudades

b) Conjunto Problema: 21 Ciudades

Table 7: Conjunto Datos Generados, 21 Ciudades

Executed Command	
python3 main.py generateSites GLPK data21 21 0 100 0 100	
OUTPUT	
GLPK data file	./data/GLPK/data21.dat
ACO data file	./data/data21.dat

1) Resultados Modelo GLPK

Table 8: Rendimiento Modelo GLPK, 21 Ciudades

Executed Command	
glpsol -m tsp.mod -d ./data/GLPK/data21.dat -o ./data/GLPK/data21-Output.dat	
OUTPUT	
GLPK Output file	./data/GLPK/data21-Output.dat
Cost	401
Route	[1,5,9,11,19,4,3,20,21,10,18,16,2,6,14,7,13,8,15,17,12]
Runtime	37.5 s

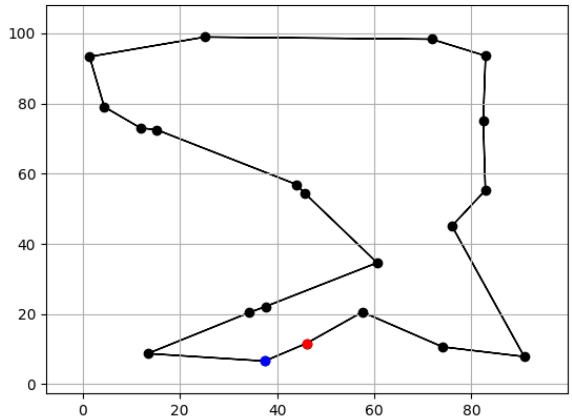

Figure 5: Ciclo Hamiltoniano Modelo GLPK, 21 Ciudades

2) Resultados Modelo ACO

Al igual que el caso anterior, se realizó diversificación en los parámetros de la colonia para encontrar los parámetros de las cinco colonias más aptas para trabajar el conjunto de datos generado de 21 Ciudades.

Para esto se realizó la generación de las 144 colonias descritas con la misma cantidad de variaciones en los rangos establecidos para el conjunto anterior de 13 Ciudades.

Table 9: Resultados Diversificación Colonias, 21 Ciudades

Executed Command								
python3 main.py solveMultipleModels GLPK data21 8 3 21 89 2 1 13 2 1 13 2 0.21 0.89 2 1 3								
OUTPUT								
Results			./data/data21-RESULTSx8.dat					
Log			./data/data21-LOGx8.dat					
cost	colony	Ants	alpha	beta	rho	Q	scheme	Runtime
401	15	43	1.06	9.43	0.32	2	0	0.66

403	66	51	1.06	9.43	0.62	2	0	0.81
408	84	51	3.98	9.43	0.32	2	0	0.85
408	138	74	3.98	9.43	0.62	2	0	1.16
410	135	74	3.98	9.43	0.32	2	0	1.14

Luego se ejecutó el modelo implementando uno de los parámetros de la mejor colonia, y usando 144 iteraciones o generaciones.

Table 10: Rendimiento Modelo ACO, 21 Ciudades

Executed Command	
python3 main.py solveModel data21 144 43 1.06 9.43 0.32 2 0	
OUTPUT	
Colony	15
Iterations	144
Cost	406
Route	[8, 17, 19, 13, 7, 14, 6, 2, 16, 18, 10, 21, 20, 3, 4, 11, 9, 5, 1, 12, 15]
Runtime	10.62 s

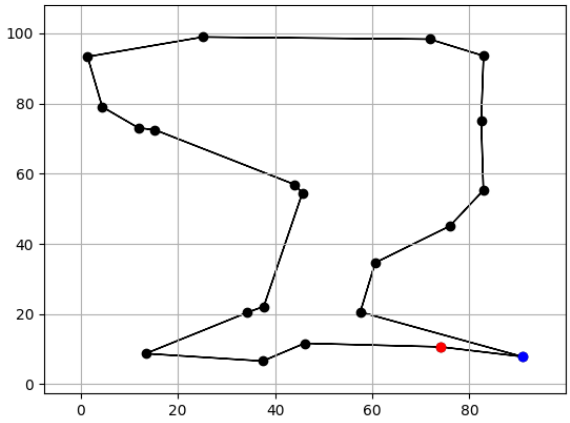

Figure 6: Figure 5: Ciclo Hamiltoniano Modelo ACO, 21 Ciudades

c) Conjunto Problema: 34 Ciudades

Table 11: Conjunto Datos Generados, 34 Ciudades

Executed Command	
python3 main.py generateSites GLPK data34 34 0 100 0 100	
OUTPUT	
GLPK data file	./data/GLPK/data34.dat
ACO data file	./data/data34.dat

1) Resultados Modelo GLPK

A partir de este caso, y en los próximos conjuntos de datos, no fue posible obtener los resultados óptimos del modelo, debido a que luego de más de un período de 4 horas, se desbordaron los recursos asignados en memoria (2048MB) para el modelo.

Table 12: Rendimiento Modelo GLPK, 34 Ciudades

Executed Command	
glpsol -m tsp.mod -d ./data/GLPK/data34.dat -o ./data/GLPK/data34-Output.dat	
OUTPUT	
GLPK Output file	None
Cost	None
Route	None
Runtime	>> 4h

2) Resultados Modelo ACO

Al igual que el caso anterior, se realizó diversificación en los parámetros de la colonia para encontrar los parámetros de las cinco colonias más aptas para trabajar el conjunto de datos generado de 34 Ciudades.

Para esto se realizó la generación de las 144 colonias descritas con la misma cantidad de variaciones en los rangos establecidos para el primer conjunto de 13 Ciudades.

Table 13: Resultados Diversificación Colonias, 34 Ciudades

Executed Command								
python3 main.py solveMultipleModels GLPK data34 8 3 21 89 2 1 13 2 1 13 2 0.21 0.89 2 1 3								
OUTPUT								
Results			./data/data34-RESULTSx8.dat					
Log			./data/data34-LOGx8.dat					
cost	colony	Ants	alpha	beta	rho	Q	scheme	Runtime
520	95	52	12.4	4.84	0.85	3	2	2
533	88	52	12.4	4.84	0.6	3	1	1.95
536	69	52	7.56	4.84	0.85	3	0	1.88
538	139	74	12.4	4.84	0.85	2	1	2.73
539	135	74	12.4	4.84	0.6	3	0	2.67

Luego se ejecutó el modelo implementando uno de los parámetros de la mejor colonia, y usando 144 iteraciones o generaciones.

Table 14: Rendimiento Modelo ACO, 34 Ciudades

Executed Command	
python3 main.py solveModel data34 144 52 12.4 4.84 0.85 3 2	
OUTPUT	
Colony	95
Iterations	144
Cost	554
Route	[13, 7, 34, 31, 10, 23, 8, 11, 14, 29, 32, 20, 3, 17, 6, 1, 19, 27, 5, 33, 2, 4, 24, 9, 22, 21, 12, 25, 26, 30, 16, 15, 18, 28]
Runtime	33.59 s

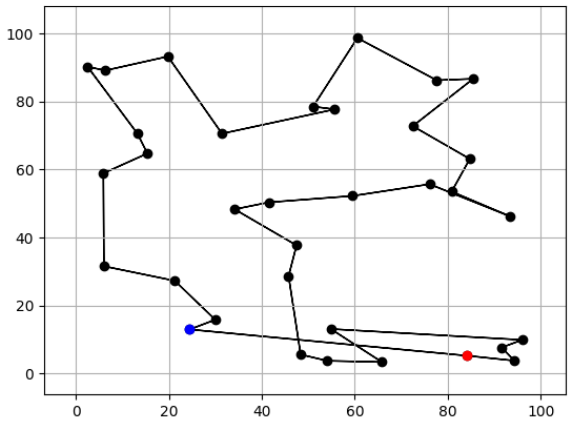

Figure 7: Ciclo Hamiltoniano Modelo ACO, 34 Ciudades

d) Conjunto Problema: 55 Ciudades

Table 15: Conjunto Datos Generados, 55 Ciudades

Executed Command	
python3 main.py generateSites GLPK data55 55 0 100 0 100	
OUTPUT	
GLPK data file	./data/GLPK/data55.dat
ACO data file	./data/data55.dat

1) Resultados Modelo ACO

Table 16: Resultados Diversificación Colonias, 55 Ciudades

Al igual que el caso anterior, se realizó la generación de las 144 colonias descritas con la misma cantidad de variaciones en los rangos establecidos para el primer conjunto.

Executed Command
python3 main.py solveMultipleModels GLPK data55 8 3 21 89 2 1 13 2 1 13 2 0.21 0.89 2 1 3

OUTPUT								
Results			./data/data55-RESULTSx8.dat					
Log			./data/data55-LOGx8.dat					
cost	colony	Ants	alpha	beta	rho	Q	scheme	Runtime
661	9	34	2.12	4.39	0.74	3	0	3.07
683	42	34	9.89	4.49	0.74	2	0	3.07
684	55	66	2.12	4.39	0.74	2	1	6
685	43	34	9.89	4.49	0.74	2	1	3.17
687	63	66	2.12	4.49	0.41	3	0	6.42

Luego se ejecutó el modelo implementando uno de los parámetros de la mejor colonia, y usando 144 iteraciones o generaciones.

Table 17: Rendimiento Modelo ACO, 55 Ciudades

Executed Command	
python3 main.py solveModel data55 144 34 2.12 4.39 0.74 3 0	
OUTPUT	
Colony	9
Iterations	144
Cost	686
Route	[55, 8, 36, 19, 25, 47, 13, 51, 52, 53, 2, 14, 16, 38, 18, 44, 20, 22, 29, 11, 32, 4, 26, 40, 43, 49, 46, 3, 34, 30, 39, 48, 9, 10, 24, 41, 1, 37, 17, 42, 33, 31, 23, 50, 15, 28, 27, 6, 12, 5, 35, 54, 21, 7, 45]
Runtime	57.03 s

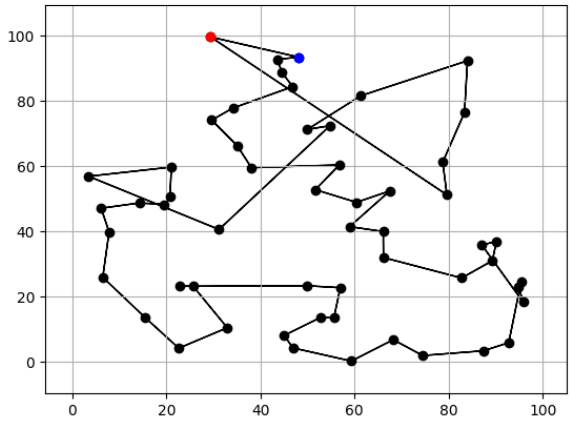

Figure 8: Ciclo Hamiltoniano Modelo ACO, 55 Ciudades

e) Conjunto Problema: 89 puntos

Table 18: Conjunto Datos Generados, 89 Ciudades

Executed Command	
python3 main.py generateSites GLPK data89 89 0 100 0 100	
OUTPUT	
GLPK data file	./data/GLPK/data89.dat
ACO data file	./data/data89.dat

1) Resultados Modelo ACO

Table 19: Resultados Diversificación Colonias, 89 Ciudades

Al igual que los casos anteriores, se realizó la generación de las 144 colonias descritas con la misma cantidad de variaciones en los rangos establecidos para el primer conjunto.

Executed Command								
python3 main.py solveMultipleModels GLPK data89 8 3 21 89 2 1 13 2 1 13 2 0.21 0.89 2 1 3								
OUTPUT								
Results			./data/data89-RESULTSx8.dat					
Log			./data/data89-LOGx8.dat					
cost	colony	Ants	alpha	beta	rho	Q	scheme	Runtime
792	112	79	1.98	9.93	0.82	2	1	19.29
798	15	31	1.98	9.93	0.82	2	0	7.47
803	118	79	1.98	9.93	0.84	2	1	19.4
809	17	31	1.98	9.93	0.82	2	2	7.41
811	60	56	1.98	9.93	0.82	2	0	13.94

Luego se ejecutó el modelo implementando uno de los parámetros de la mejor colonia, y usando 144 iteraciones o generaciones.

Table 20: Rendimiento Modelo ACO, 89 Ciudades

Executed Command	
python3 main.py solveModel data89 144 79 1.98 9.93 0.82 2 1	
OUTPUT	
Colony	112
Iterations	144
Cost	793
Route	[83, 47, 9, 30, 22, 63, 32, 67, 44, 61, 88, 86, 62, 31, 85, 1, 75, 28, 4, 41, 29, 38, 71, 11, 18, 49, 66, 81, 56, 69, 7, 5, 51, 2, 70, 17, 23, 42, 89, 27, 13, 58, 53, 48, 8, 15, 55, 26, 37, 35, 19, 54, 84, 45, 43, 6, 64, 60, 68, 73, 79, 10, 65, 16, 33, 20, 46, 87, 78, 12, 21, 14, 24, 72, 40, 74, 59, 77, 34, 82, 36, 3, 80, 50, 39, 52, 57, 76, 25]
Runtime	360.58s

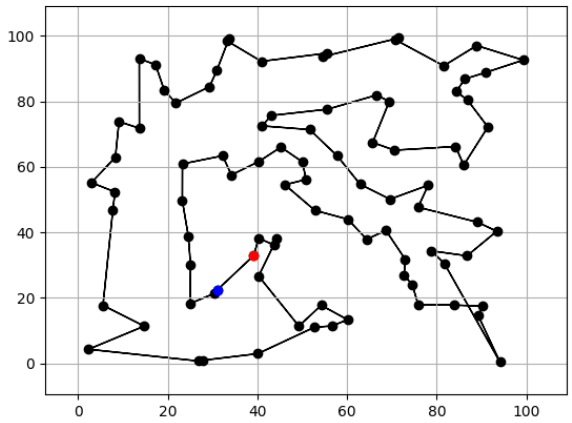

Figure 9: Ciclo Hamiltoniano Modelo ACO, 89 Ciudades

VI. Conclusiones

En base a los resultados obtenidos en el apartado anterior, podemos observar que para cada conjunto de datos del problema, existe una cantidad de parámetros que aportan mayor peso en la obtención de la mejor solución, y dependen en concreto de la forma en cómo se encuentra distribuidos los datos del conjunto.

Al observar los resultados de diversificación de colonias sobre el conjunto de datos para 89 Ciudades, se pudo observar que los parámetros con mayor peso en la obtención de la mejor solución fue el factor alpha, beta, rho y Q.

En general, para cualquier conjunto de datos, los parámetros que tienen mayor peso en la obtención de la mejor solución son: el número de hormigas de la colonia, el factor rho de evaporación de la feromona; y por último, el factor alfa y beta, referente a la influencia de la feromona y su línea de vista.

Sin embargo, al generar las 144 colonias sobre el conjunto de 34 ciudades se observó que la mejor colonia entregó un costo de 520 para 8 iteraciones o generaciones, y un costo de 554 al ejecutar la misma colonia realizando 144 iteraciones o generaciones, lo cual sólo es posible si asumimos que también existe un ruido asociado a los parámetros de las mejores colonias, debido a la aleatoriedad del modelo al momento de

que cada hormiga de la colonia selecciona la próxima ciudad, lo que puede obligar a establecer nuevas estrategias en la elección de los mejores parámetros de la colonia para un conjunto de datos específico.

Por último, se pudo observar la gran capacidad de recursos de memoria y procesamiento necesarios para poder resolver el problema usando el modelo óptimo GLPK. A pesar de que en el primer conjunto de 13 Ciudades se obtuvo un rendimiento mucho mayor al modelo ACO, su rendimiento cayó drásticamente a medida que se incrementó la entrada del problema.

Por tanto, resulta eficaz la implementación de un modelo metaheurístico basado en el comportamiento social de las especies que compiten por comida para la obtención de la solución lo suficientemente buena para el Problema del Agente Viajero (TSP) y en un tiempo de ejecución razonable.

VII. ANEXOS

A continuación se puede observar la información general para el uso del modelo ACO-TSP.

a) ACO-TSP

Resuelve el problema del Agente Viajero usando Optimización por Colonia de Hormigas a través del lenguaje e intérprete de programación Python3.

b) Requerimientos

- Python3
- matplotlib

c) Repositorio

El modelo desarrollado puede ser clonado a través del Repositorio de GitHub, a través del siguiente comando.

- git clone https://github.com/JohanyCarmona/ACO-TSP.git

d) Uso

Ejecutar el comando bash: ‘python3 main.py’ para ver la ayuda general del programa.

- generateSites [filename] [totalSites] [longitudeRangeMin] [longitudeRangeMax] [latitudeRangeMin] [latitudeRangeMax]

  Permite generar un archivo de texto plano con un conjunto determinado de ciudades distribuidas aleatoriamente en un rango determinado por el usuario.

- generateSites GLPK [filename] [totalSites] [longitudeRangeMin] [longitudeRangeMax] [latitudeRangeMin] [latitudeRangeMax]

  Permite generar dos archivos de texto plano con un conjunto determinado de ciudades distribuidas aleatoriamente en un rango determinado por el usuario para usar en el modelo ACO y GLPK.

  El archivo de texto plano con el formato GLPK se genera en el subdirectorio de datos: ./data/GLPK/

- solveModel [filename] [iterations] [totalAnts] [alpha] [beta] [rho] [Q] [scheme]

  Permite resolver el conjunto determinado de ciudades proveniente del archivo de texto plano usando los parámetros de la colonia de hormigas especificada por el usuario.

- solveMultipleModels [filename] [iterationsPerColony] [totalAntsAlterations] [totalAntsMin] [totalAntsMax] [alphaAlterations] [alphaMin] [alphaMax] [betaAlterations] [betaMin] [betaMax] [rhoAlterations] [rhoMin] [rhoMax] [QAlterations] [QMin] [Qmax]

  Permite resolver el conjunto determinado de ciudades proveniente del archivo de texto plano usando múltiples variaciones en los parámetros de la colonia de hormigas en base a los rangos especificados por el usuario.

  Se recomienda generar un número de iteraciones o generaciones por colonia relativamente bajo con el fin de encontrar los parámetros de las mejores colonias que se utilizarán para intensificar la búsqueda de la mejor solución.

- plotRoute [filename] [indexSite0] [indexSite1] [indexSite3] ... [indexSiteN]

  Permite generar de forma gráfica el camino, ruta o ciclo hamiltoniano especificado por el usuario para el conjunto determinado de ciudades proveniente del archivo de texto plano, para comparar los resultados generados externamente por el modelo óptimo GLPK, con los resultados del modelo ACO obtenido.

VIII. Referencias

[1] Yang J., Shi X., Marchese M., Liang Y., Noviembre 2008 ,«An ant colony optimization method for generalized TSP problem» [En línea]. Available: https://www.sciencedirect.com/science/article/pii/S1002007108002736

[2] Sancho Caparrini, F. Noviembre 2018, «Algoritmo de hormigas y el problema del viajante» [En línea]. Available: www.cs.us.es/~fsancho/?e=71

[3] Robles Algarín, C. A. Noviembre 2010, «Optimización por Colonia de Hormigas: Aplicaciones y Tendencias», en Revista Ingeniería Solidaria, vol. 6, núm. 10, pp.83-89

[4] Zhang, R. Abril 2017, «Solve TSP using Ant Colony Optimization in Python 3» [En línea]. Available GitHub: https://github.com/ppoffice/ant-colony-tsp